Part 15: GPYtorch and Custom Kernels

March 8, 2021

This post will be focused on using *GPYtorch* tools to make a custom kernel matrix for use in Gaussian processes. We are using this tool because there is an ecosystem of python *torch* tools were hyperparameter optimization, constraints, priors, and integration are built into the code, providing easy access to tools much more powerful than what I can code in a week.

1 GP Regression using *GPYtorch*

We start by modeling $y = sin(2\pi)+\epsilon$ using a GP with constant $\mu(x)$ and $\Sigma(x, x') = \sigma_f^2 \times \Sigma_{RBF}(x, x', \lambda)$, which is the common GP with radial basis function covariance and scaling parameter. Using *GPYtorch* we use a standard Gaussian likelihood (which has been shown in other posts) to train the model using autograd functions to construct gradients for use in ADAM optimizer of hyperparameters. We have already gone over stochastic gradient descent and other optimizers for random functions so I won't go into the details (we may do a post on how *GPYtorch* does this in the future). This is also the simplest GP model we usually consider because it only has one length-scale parameter, one output-scale parameter, and one noise parameter. With some random data N and dimension $p = 1$ we get nice regression.

2 Custom Hyperparameters

If we want to implement custom hyperparameters that needs to be done in our *CustomKernel* class. I have coded a generic *lengthscale* and *outputscale* attached to the *self* provided in python, and shared it in the *forward* subfunction. This subfunction does the heavy lifting in the code using *numpy* and is re-converted to tensors. For the interested, I have added (i) a generic RBF kernel, (ii) a multi-fidelity kernel, and (iii) a linear kernel.

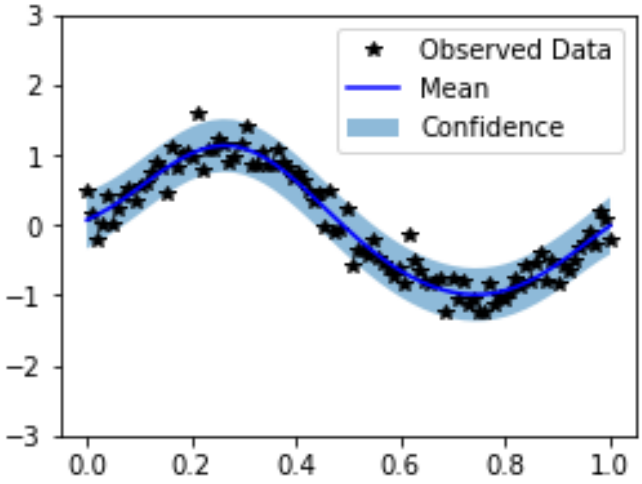
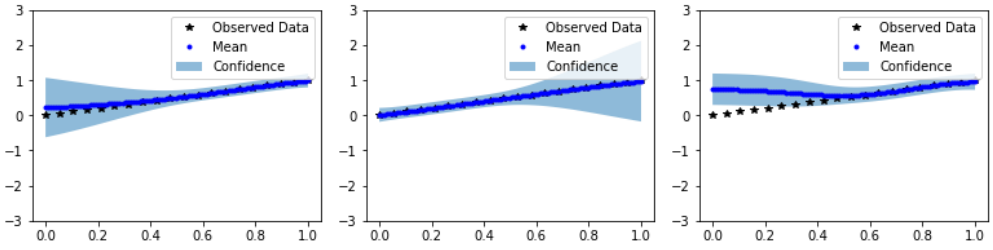


Figure 1: Sinusoidal Regression using *GPYtorch* GP

3 Custom Kernels

The interesting custom kernel implemented here is a multi-fidelity kernel. See previous posts for multi-fidelity Bayesian optimization and modeling. The problem here will just be noiseless $Y = X$ with regularly spaced input points. For the 'true' fidelity we'll use $N = 5$ points $[0.5, 1]$ and for the lower fidelity we'll use $N = 5$ points $[0, 0.5]$. If it works, the multi-fidelity model will be able to use the lower fidelity points to better predict the line, whereas the normal GP, not having access to those points, cannot. From Figure 2 we see that this is the case.



(a) Fidelity 0 (b) Fidelity 1 (c) Regular GP

Figure 2: Custom Kernel Experiments | custom multi-fidelity kernel using $N = 10$ data points vs regular kernel using $N = 5$ data points

That's pretty much what I wanted to talk about here. We didn't learn about any new cool models only some code stuff so the juicy details are in the python

code. Next week we'll do more fun modeling stuff with GPs using linear basis functions.